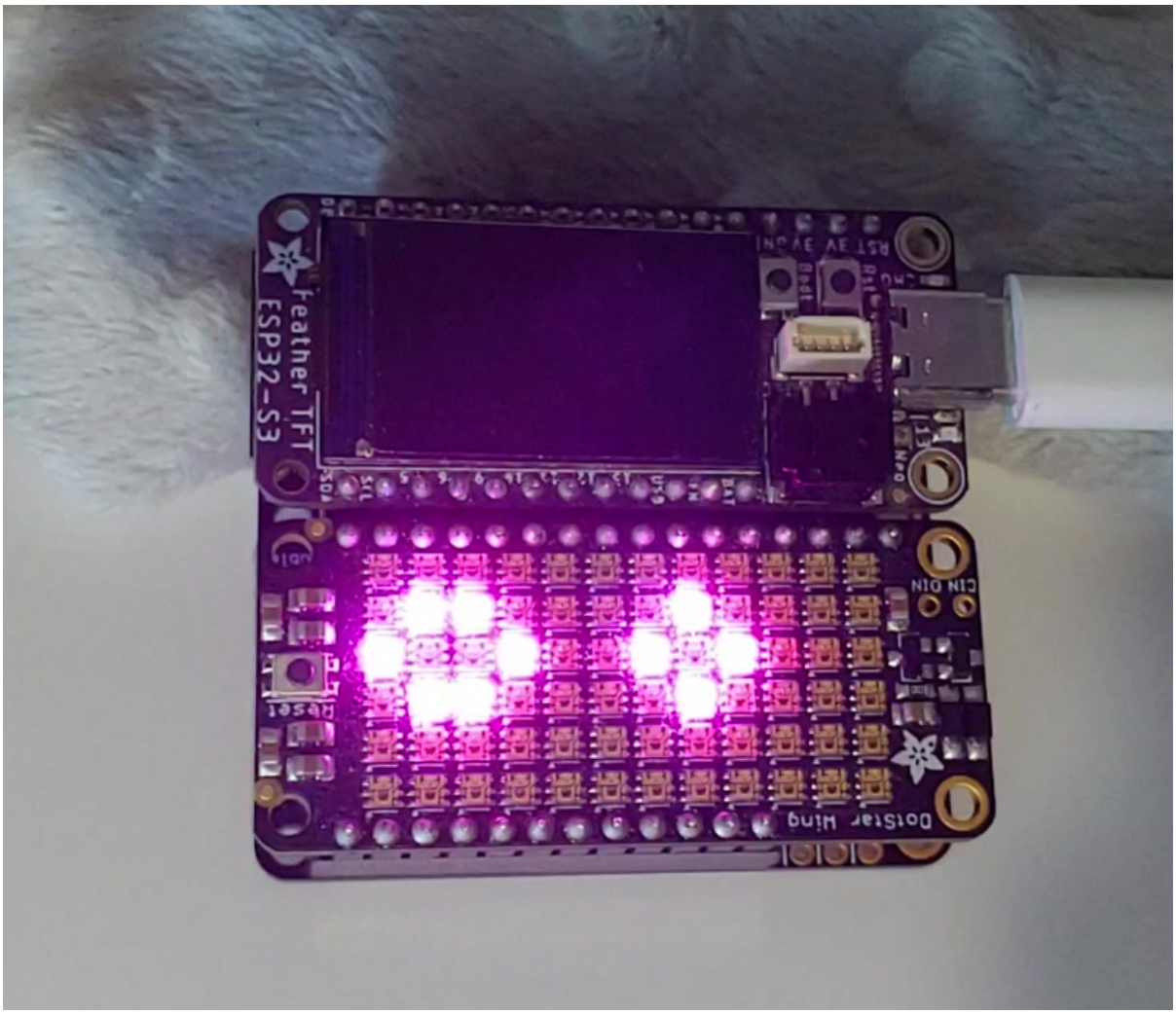
Creating Custom LED Animations

Created by Tim C



https://learn.adafruit.com/creating-custom-led-animations

Last updated on 2024-11-06 01:10:49 PM EST

Table of Contents

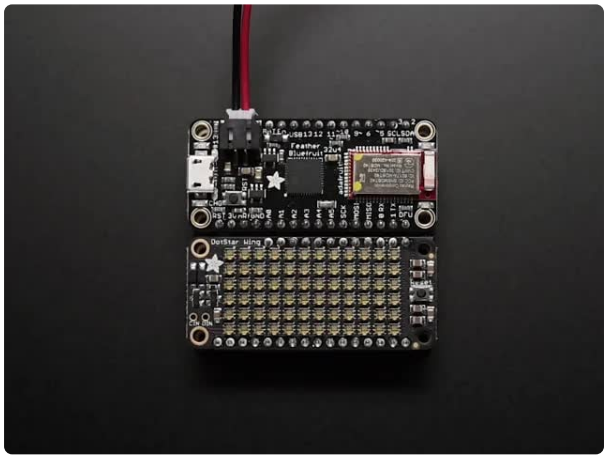
Overview

The Adafruit LED Animation Library (https://adafru.it/LZF) provides a quick and easy way to get strands or arrays of RGB LEDs blinking in concert. There are several dazzling animations built into the library that are ready to use and can be adapted to different behaviors with configuration arguments. But if none of those built-in animations fits the exact vision you have for your next great project, the LED Animation library also provides a framework for you to create your own custom animations.

This will require more than just a small bit of configuration, but is well within reach if you've slightly comfortable with Python. This guide will explain what goes into making custom animations and show some examples to illustrate different aspects of them.

Parts

Any Neopixel or DotStar compatible RGB LEDs.



Adafruit DotStar FeatherWing - 6 x 12 RGB LEDs

A Feather board without ambition is a Feather board without FeatherWings! This is the DotStar FeatherWing, a 6x12 RGB LED Add-on For All Feather Boards! Using...

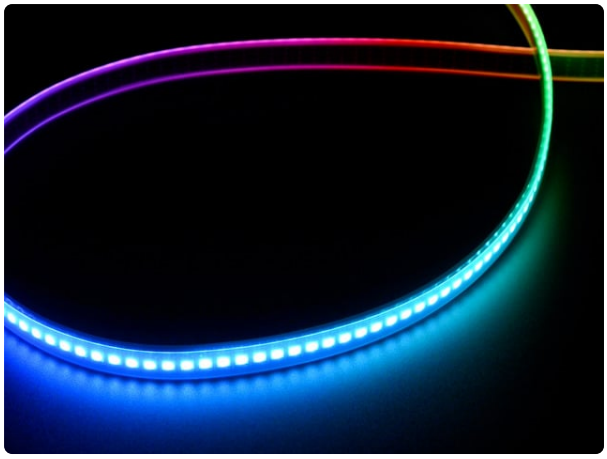
https://www.adafruit.com/product/3449

Adafruit NeoPixel LED Dots Strand - 20 LEDs at 2" Pitch

Attaching NeoPixel strips to your costume can be a struggle as the flexible PCBs can crack when bent too much. So how to add little dots of color? Use these stranded NeoPixel dots!...

https://www.adafruit.com/product/3630

[Adafruit NeoPixel LED Dots Strand - 20 LED 4" Pitch]
Attaching NeoPixel strips to your costume can be a struggle as the flexible PCBs can crack when bent too much. So how to add little dots of color? Use these stranded NeoPixel dots!...
https://www.adafruit.com/product/3631



[Adafruit DotStar Digital LED Strip - Black 144 LED/m - 0.5 Meter]
Move over NeoPixels, there's a new LED strip in town! These fancy new DotStar LED strips are a great upgrade for people who have loved and used NeoPixel strips for a few years but...
https://www.adafruit.com/product/2328

[NeoPixel FeatherWing - 4x8 RGB LED Add-on For All Feather Boards]
A Feather board without ambition is a Feather board without FeatherWings! This is the NeoPixel FeatherWing, a 4x8 RGB LED Add-on For All Feather...
https://www.adafruit.com/product/2945

Extend Animation Class

If you aren't already familiar with the `adafruit_led_animation` library, then you should head over to the primary guide for that library (https://adafru.it/LZF) to learn the basics about it first and practice using some of the built-in animations on your device.

Once you've done that then you can come back here and have an easier time understanding the class that we'll be working with, and how to instantiate and use it from user code in the **code.py** file.

The Minimum Requirements

The main thing that we need to do in order to create our custom animations is to define a new class that extends `adafruit_led_animation.Animation`. In our class we'll implement a few key functions and fill in the behavior that we want for the desired blinking.

To start, create a new Python file on your **CIRCUITPY** drive, you can place it at the root of the drive or inside of the **lib** folder. Try to give at a name that describes what the animation is, hopefully something that your future self will recognize and remember the meaning of. I'll use **demo_animation.py**, yours should be specific to your project or animation effect.

demo_animation.py

```
from adafruit_led_animation.animation import Animation

# class definiation named similarly to the filename,
# extend the Animation class from the library
class DemoAnimation(Animation):

    # init function definition called by user
    # code to create an instance of the class
    def __init__(self, pixel_object, speed, color):

        # call the init function for the super class Animation,
        # pass through all of the arguments that came to us
        super().__init__(pixel_object, speed, color)

        # initialize any other variables you need for your
        # custom animation here
        # i.e. self.my_var = "something"

    # draw function will get called from animate when the time
    # comes for it based on the configured speed
    def draw(self):
        # do something with self.pixel_object
        self.pixel_object.fill(self.color)
```

That is the bare minimum required for your custom `Animation`. It won't produce anything particularly interesting or blinky, but it can be initialized and you can call `animate()` on it in order to test.

Lets break down what the different parts of this code do:

`class DemoAnimation(Animation):`

This line defines our class, we get to name it whatever want, I've used `DemoAnimation`, yours should try to describe your own animation effect. The

`(Animation)` in parentheses means that our class will extend the `Animation` class from the `adafruit_led_animation` library. Extending that class will give us access to all of the behavior that is in the `Animation` class without us having to write it inside of our own class. We can rely on that `Animation` super class to provide most of what we need, we'll be able to just focus on the specific ways we want the RGB lights to turn on and off, and leave the rest up to it.

```
def __init__(self, pixel_object, speed, color):
```

This starts the definition for our `__init__` function which is what will be called when someone makes a new instance of our class. It accepts the same arguments as the super class `Animation`, namely `pixel_objects`, `speed`, and `color`. If we wanted to, we could add additional arguments for our specific custom animation.

```
super().__init__(pixel_object, speed, color)
```

This calls the `__init__` function for the super class `Animation`, it will do all of the internal setup required for it to provide it's behavior. We pass through the same argument values that were passed to our class's `__init__` function.

```
def draw(self):
```

This starts the definition for the `draw()` function. It's the place where we will put the code that controls when and how the lights turn on and off. The `Animation` super class that we are using will take care of checking how much time has passed and calling `draw()` at the appropriate times based on the `speed` configuration. Inside of `draw()` we can access `self.pixel_object` and use it to manipulate the RGB pixels.

code.py

```
import board
import neopixel
from demo_animation import DemoAnimation

# Update to match the pin connected to your NeoPixels
pixel_pin = board.D10
# Update to match the number of NeoPixels you have connected
pixel_num = 32

pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.1, auto_write=False)

demo_anim = DemoAnimation(pixels, 0.5, (255, 0, 255))

while True:
    demo_anim.animate()
```

If you run this **code.py** file using the `DemoAnimation` class shown above it will simply turn all of the LEDs on to a pink color (assuming they are RGB color order). To do something more interesting we would expand upon the code inside of the `draw()`

function to make it do other stuff like turning LEDs off sometimes, or setting some to a color and others to different colors, or perhaps changing the color they're showing with each successive call to `draw()`. There are examples on the following pages that illustrate different possibles ideas and are thoroughly commented to explain how they work.

Optional Functionality

The rest of the features of the `Animation` class are optional. Certain animations or projects might find some usage for them, but others may be able to work perfectly fine with only the pieces shown above.

`on_cycle_complete_supported`

The cycle complete receiver is a way for user code to get a callback whenever a full cycle of the animation is complete. This allows the user code to insert some customized functionality when that time comes. For example maybe you have a project that wants to play a short audio effect each time the LED animation reaches its loop point. In order to be able to use the cycle complete receiver the specific `Animation` class in use must set `on_cycle_complete_supported = True`. It must also set `self.cycle_complete = True` somewhere inside of the `draw()` function at the appropriate time based on how it's cycle works. The logic used for determining when the cycle is complete is up to the specific usages within the custom `Animation`.

```
class DemoAnimation(Animation):
    on_cycle_complete_supported = True
    def draw(self):
    if <your cycle logic here>:
        self.cycle_complete = True

    # ... rest of the Animation definition ...
```

code.py

```
def play_sound():
    print("cycle complete play the sound")
    # ... your audio code here ...

demo_anim = DemoAnimation(pixelmap_up, speed=.1, color=AMBER)
demo_anim.add_cycle_complete_receiver(play_sound)

while True:
    demo_anim.animate()
```

`reset()`

The `reset()` function can be overridden by subclasses of `Animation` in order to insert some customized behavior that happens at the end of a full cycle of the `Animation` and before the next cycle begins. A good example of this is the built-in `Comet` animation which has an overridden `reset()` function that carries out the required behavior for it's `reverse` and `ring` features.

If your custom Animation has some behavior it needs to do when the animation loops then you need to do two things in your custom class: 1) override the `reset()` function to put your behavior inside of it, and 2) add some code into the `draw()` function that calls `reset()` at the appropriate time.

> Reset is a similar concept to the cycle complete receiver discussed above. The main difference is reset() is meant for use internally within the Animation. While the cycle complete reciever is intended for user code that is outside of our custom Animation class, but is going to make an instance of our class and call animate() on it.

```
class DemoAnimation(Animation):

    def __init__(self, pixel_object, speed, color,):
        super().__init__( pixel_object, speed, color)
        self.index = 0

    def reset(self):
        self.pixel_object.fill((0, 0, 0))
        self.index = 0

    def draw(self):
        if self.index >= len(self.pixel_object):
            self.reset()
        self.pixel_object[self.index] = self.color
        self.index += 1
```

This example DemoAnimation uses a custom variable `index` to store the current index within the full strip of pixels. The overridden `reset()` function sets `index` back to zero and turns off all of the pixels. The `draw()` function calls `reset()` when the index has reached to the length of the `pixel_object`. The result is the animation will turn on one additional pixel per frame. Once all pixels are on, they will be turned off and cycle will repeat back at the beginning of the strip.

`after_draw()`

The `after_draw()` function can be overridden by subclasses of `Animation` to insert customized behavior that will occur after each animation frame. For example the built-in `Sparkle` animation overrides the `after_draw()` function to set some of the pixels to a dimmer color in order to create the sparkling visual effect.

`after_draw()` will get called automatically at the conclusion of the `draw()` function, you do not need to manually call it yourself inside `draw()` or anywhere else.

```
class DemoAnimation(Animation):

    def __init__(self, pixel_object, speed, color,):
        super().__init__( pixel_object, speed, color)
        self.drawn_pixels = []

    def draw(self):
        random_index = random.randint(0, len(self.pixel_object)-1)
        self.pixel_object[random_index] = self.color
        self.drawn_pixels.append(random_index)

    def after_draw(self):
        if len(self.drawn_pixels) > 5:
            oldest_index = self.drawn_pixels.pop(0)
            self.pixel_object[oldest_index] = (0,0,0)
```

This example `DemoAnimation` turns on random LEDs one at a time until there are 5 LEDs on. After 5 LEDs are on, the `after_draw()` function is used to turn off the LED that has been on the longest. The result is that random LEDs turn on, and stay on for 5 frames of the animation and then turn off. It uses the custom variable `drawn_pixels` to store a list of indexes of the LEDs that are currently on.

With all of the tools discussed on this page we have everything we need to create dazzling and delightful animations. On the next page we'll look at some some other specific examples of custom animations.

Custom Animation Examples

There are more example animations on this page that you can look at in order to get ideas about how you can manipulate the LEDs from your custom `Animation`. Another really great place to look is the existing built-in Animations from the adafruit_led_animation library. (https://adafru.it/1a9A) The examples in this guide are commented thoroughly detailing the purpose of each line of code. If you'd like to try these on your own device simply click the 'Download Project Bundle' button at the top of the code files. It will download the custom `Animation` class, a **code.py** file that uses it, and all required libraries it needs to run.

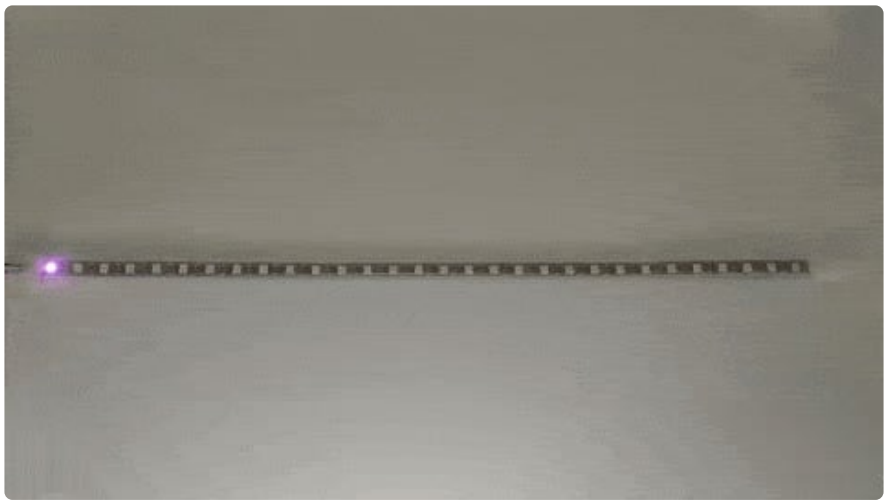
Starting from existing animations

When you'd like to create your own custom Animation it's helpful if you can find an existing animation that is similar in some way to the animation you have in mind. If there is one with any similarities then you can copy it's code to start yours from instead of needing to start with an absolute "blank canvas". For instance, if you know

that you want a rainbow animation you can re-use the internal color wheel generator from one of the built-in animations.

Sweep Animation

The `SweepAnimation` will sweep across the strand, turning 1 pixel on with each animation frame. Once all of the pixels are on it will then sweep across again the same direction turning them back off. It uses custom variables to keep track of whether it's currently sweeping on or off, as well as the current index within the strand.

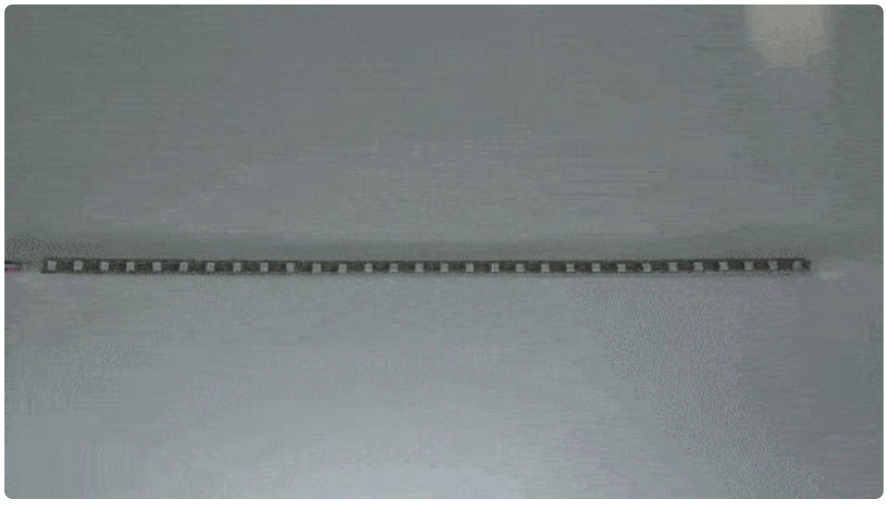


```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
"""
SweepAnimation helper class
"""
from adafruit_led_animation.animation import Animation


class SweepAnimation(Animation):

    def __init__(self, pixel_object, speed, color):
        """
        Sweeps across the strand lighting up one pixel at a time.
        Once the full strand is lit, sweeps across again turning off
        each pixel one at a time.

        :param pixel_object: The initialized pixel object
        :param speed: The speed to run the animation
        :param color: The color the pixels will be lit up.
        """

        # Call super class initialization
        super().__init__(pixel_object, speed, color)

        # custom variable to store the current step of the animation
        self.current_step = 0

        # one step per pixel
        self.last_step = len(pixel_object)

        # boolean indicating whether we're currently sweeping LEDs on or off
        self.sweeping_on = True
```

```
        self.cycle_complete = False

    # This animation supports the cycle complete callback
    on_cycle_complete_supported = True

    def draw(self):
        """
        Display the current frame of the animation

        :return: None
        """
        if self.sweeping_on:
            # Turn on the next LED
            self.pixel_object[self.current_step] = self.color
        else:  # sweeping off
            # Turn off the next LED
            self.pixel_object[self.current_step] = 0x000000

        # increment the current step variable
        self.current_step += 1

        # if we've reached the last step
        if self.current_step >= self.last_step:

            # if we are currently sweeping off
            if not self.sweeping_on:
                # signal that the cycle is complete
                self.cycle_complete = True

            # reset the step variable to 0
            self.current_step = 0

            # flop sweeping on/off indicator variable
            self.sweeping_on = not self.sweeping_on
```

Zipper Animation

The `ZipperAnimation` will start lighting up every other LED from both ends of the strand, passing each other in the middle and resulting in the full strand being lit at the end of the cycle. It uses custom variables to keep track of the current step within the animation. The `reset()` is used to turn off the LEDs and reset the variables back to their initial values.

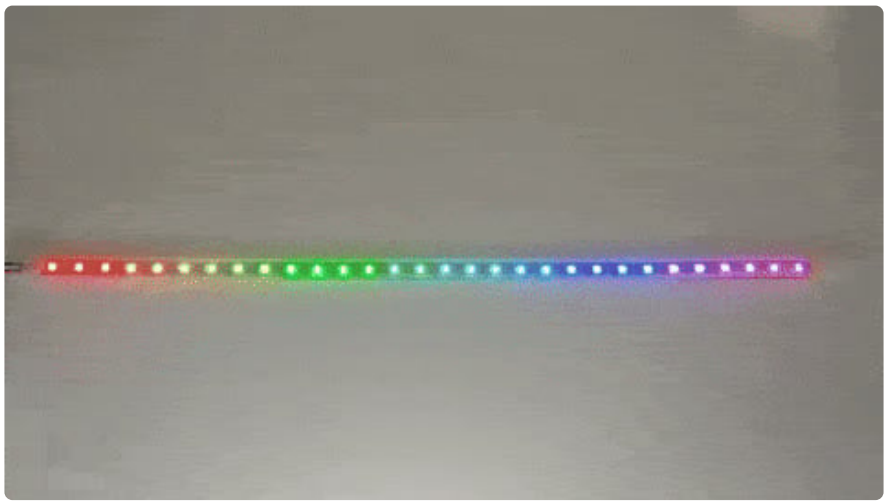


```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
"""
ZipperAnimation helper class
"""
from adafruit_led_animation.animation import Animation


class ZipperAnimation(Animation):

    def __init__(self, pixel_object, speed, color, alternate_color=None):
        """
        Lights up every other LED from each ends of the strand, passing each
        other in the middle and resulting in the full strand being lit at the
        end of the cycle.

        :param pixel_object: The initialized pixel object
        :param speed: The speed to run the animation
        :param color: The color the pixels will be lit up.
        """

        # Call super class initialization
        super().__init__(pixel_object, speed, color)

        # if alternate color is None then use single color
        if alternate_color is None:
            self.alternate_color = color
        else:
            self.alternate_color = alternate_color

        # custom variable to store the current step of the animation
        self.current_step = 0

        # We're lighting up every other LED, so we have half the strand
        # length in steps.
        self.last_step = len(pixel_object) // 2

        self.cycle_complete = False

    # This animation supports the cycle complete callback
    on_cycle_complete_supported = True

    def draw(self):
        """
        Display the current frame of the animation

        :return: None
        """

        # Use try/except to ignore indexes outside the strand
        try:
            # Turn on 1 even indexed pixel starting from the start of the strand
            self.pixel_object[self.current_step * 2] = self.color

            # Turn on 1 odd indexed pixel starting from the end of the strand
            self.pixel_object[-(self.current_step * 2) - 1] = self.alternate_color
        except IndexError:
            pass

        # increment the current step variable
        self.current_step += 1

        # if we've reached the last step
        if self.current_step > self.last_step:
            # signal that the cycle is complete
            self.cycle_complete = True
```

```
            # call internal reset() function
            self.reset()

    def reset(self):
        """
        Turns all the LEDs off and resets the current step variable to 0
        :return: None
        """
        # turn LEDs off
        self.pixel_object.fill(0x000000)

        # reset current step variable
        self.current_step = 0
```

RainbowSweep Animation

The `RainbowSweepAnimation` shimmers the whole strand with a rainbow and then sweeps across it with another specified color. You can use `BLACK` or `0x000000` to turn LEDs off during the sweep. This animation uses a generator function internally to iterate through colors of the rainbow with help some colorwheel helpers. Many of the built-in Rainbow animation use this technique.



```
# SPDX-FileCopyrightText: 2024 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Adapted From `adafruit_led_animation.animation.rainbow`
"""

from adafruit_led_animation.animation import Animation
from adafruit_led_animation.color import colorwheel
from adafruit_led_animation import MS_PER_SECOND, monotonic_ms


class RainbowSweepAnimation(Animation):
    """
    The classic rainbow color wheel that gets swept across by another specified
color.

    :param pixel_object: The initialised LED object.
    :param float speed: Animation refresh rate in seconds, e.g. ``0.1``.
    :param float sweep_speed: How long in seconds to wait between sweep steps
    :param float period: Period to cycle the rainbow over in seconds.  Default 1.
```

```
    :param sweep_direction: which way to sweep across the rainbow. Must be one of
      DIRECTION_START_TO_END or DIRECTION_END_TO_START
    :param str name: Name of animation (optional, useful for sequences and
debugging).

    """

    # constants to represent the different directions
    DIRECTION_START_TO_END = 0
    DIRECTION_END_TO_START = 1
    # pylint: disable=too-many-arguments
    def __init__(
        self, pixel_object, speed, color, sweep_speed=0.3, period=1,
            name=None, sweep_direction=DIRECTION_START_TO_END
    ):
        super().__init__(pixel_object, speed, color, name=name)
        self._period = period
        # internal var step used inside of color generator
        self._step = 256 // len(pixel_object)

        # internal var wheel_index used inside of color generator
        self._wheel_index = 0

        # instance of the generator
        self._generator = self._color_wheel_generator()

        # convert swap speed from seconds to ms and store it
        self._sweep_speed = sweep_speed * 1000

        # set the initial sweep index
        self.sweep_index = len(pixel_object)

        # internal variable to store the timestamp of when a sweep step occurs
        self._last_sweep_time = 0

        # store the direction argument
        self.direction = sweep_direction

    # this animation supports on cycle complete callbacks
    on_cycle_complete_supported = True

    def _color_wheel_generator(self):
        # convert period to ms
        period = int(self._period * MS_PER_SECOND)

        # how many pixels in the strand
        num_pixels = len(self.pixel_object)

        # current timestamp
        last_update = monotonic_ms()

        cycle_position = 0
        last_pos = 0
        while True:
            cycle_completed = False
            # time vars
            now = monotonic_ms()
            time_since_last_draw = now - last_update
            last_update = now

            # cycle position vars
            pos = cycle_position = (cycle_position + time_since_last_draw) % period

            # if it's time to signal cycle complete
            if pos < last_pos:
                cycle_completed = True

            # update position var for next iteration
            last_pos = pos
```

```
            # calculate wheel_index
            wheel_index = int((pos / period) * 256)

            # set all pixels to their color based on the wheel color and step
            self.pixel_object[:] = [
                colorwheel(((i * self._step) + wheel_index) % 255) for i in
range(num_pixels)
            ]

            # if it's time for a sweep step
            if self._last_sweep_time + self._sweep_speed <= now:

                # udpate sweep timestamp
                self._last_sweep_time = now

                # decrement the sweep index
                self.sweep_index -= 1

                # if it's finished the last step
                if self.sweep_index == -1:
                    # reset it to the number of pixels in the strand
                    self.sweep_index = len(self.pixel_object)

            # if end to start direction
            if self.direction == self.DIRECTION_END_TO_START:
                # set the current pixels at the end of the strand to the specified
color
                self.pixel_object[self.sweep_index:] = (
                        [self.color] * (len(self.pixel_object) - self.sweep_index))

            # if start to end direction
            elif self.direction == self.DIRECTION_START_TO_END:
                # set the pixels at the begining of the strand to the specified
color
                inverse_index = len(self.pixel_object) - self.sweep_index
                self.pixel_object[:inverse_index] = [self.color] * (inverse_index)

            # update the wheel index
            self._wheel_index = wheel_index

            # signal cycle complete if it's time
            if cycle_completed:
                self.cycle_complete = True
            yield


    def draw(self):
        """
        draw the current frame of the animation
        :return:
        """
        next(self._generator)

    def reset(self):
        """
        Resets the animation.
        """
        self._generator = self._color_wheel_generator()
```

All in Sequence

The following **code.py** file will play all 3 animations one after another in an `AnimationSequence`. Click the download project button to try it out. Remember to

update the pin referenced by the code to match the pin your Neopixels are connected to.

```
# SPDX-FileCopyrightText: 2024 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import board

import neopixel
from adafruit_led_animation.color import PINK, JADE
from adafruit_led_animation.sequence import AnimationSequence

from rainbowsweep import RainbowSweepAnimation
from sweep import SweepAnimation
from zipper import ZipperAnimation

# Update to match the pin connected to your NeoPixels
pixel_pin = board.A1
# Update to match the number of NeoPixels you have connected
pixel_num = 30

# initialize the neopixels. Change out for dotstars if needed.
pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.02, auto_write=False)

# initialize the animations
sweep = SweepAnimation(pixels, speed=0.05, color=PINK)

zipper = ZipperAnimation(pixels, speed=0.1, color=PINK, alternate_color=JADE)

rainbowsweep = RainbowSweepAnimation(pixels, speed=0.05, color=0x000000,
sweep_speed=0.1,

sweep_direction=RainbowSweepAnimation.DIRECTION_END_TO_START)

# sequence to play them all one after another
animations = AnimationSequence(
    sweep, zipper, rainbowsweep, advance_interval=6, auto_clear=True
)

while True:
    animations.animate()
```

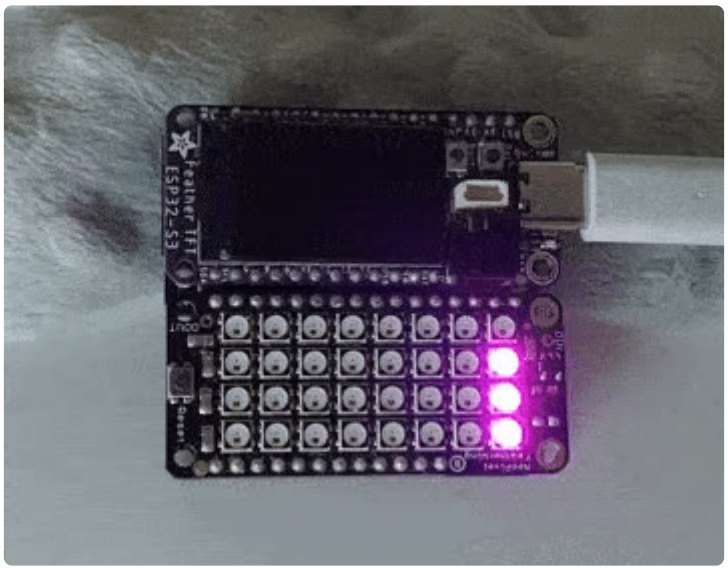
2D Grid Examples

The `adafruit_led_animation` library contains a built-in helper class (https://adafru.it/1a9B) for dealing with 2D Grids of pixels, `PixelGrid`. We can use this class inside of our custom Animation to be able to refer to the pixels with x, y coordinates in a grid rather than just an index within a strand like normal. This will only make sense to use if your pixels are also physically arranged in a grid, the Neopixel FeatherWing (http://adafru.it/2945) or Dotstar FeatherWing (http://adafru.it/3449) provide an easy to use grid to run them on.

Snake Animation

The `SnakeAnimation` will make a snake of the specified size and color that will slither around the grid in random directions. If the snake gets stuck in a corner a new one will spawn and carry on slithering around. The current snake segment locations

are stored in a list variable. During each animation step the snake will decide randomly whether to continue the direction it's going, or to change to a different random direction. Much of the movement logic is delegated into helper functions `_can_move()` and `_choose_direction()`.



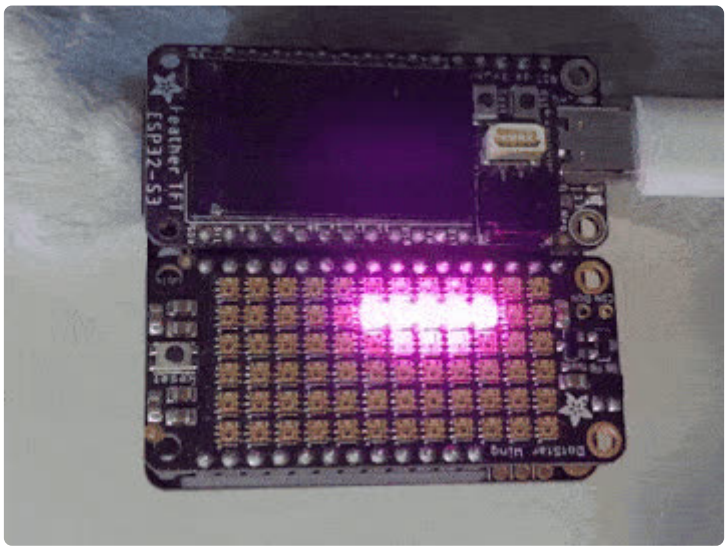
```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
"""
SnakeAnimation helper class
"""
import random
from micropython import const

from adafruit_led_animation.animation import Animation
from adafruit_led_animation.grid import PixelGrid, HORIZONTAL



class SnakeAnimation(Animation):
    UP = const(0x00)
    DOWN = const(0x01)
    LEFT = const(0x02)
    RIGHT = const(0x03)
    ALL_DIRECTIONS = [UP, DOWN, LEFT, RIGHT]
    DIRECTION_OFFSETS = {
        DOWN: (0, 1),
        UP: (0, -1),
        RIGHT: (1, 0),
        LEFT: (-1, 0)
    }

    def __init__(self, pixel_object, speed, color, width, height, snake_length=3):
        """
        Renders a snake that slithers around the 2D grid of pixels
        """
        super().__init__(pixel_object, speed, color)

        # how many segments the snake will have
        self.snake_length = snake_length

        # create a PixelGrid helper to access our strand as a 2D grid
        self.pixel_grid = PixelGrid(pixel_object, width, height,
                                    orientation=HORIZONTAL, alternating=False)
```

```
        # size variables
        self.width = width
        self.height = height

        # list that will hold locations of snake segments
        self.snake_pixels = []

        self.direction = None

        # initialize the snake
        self._new_snake()

    def _clear_snake(self):
        """
        Clear the snake segments and turn off all pixels
        """
        while len(self.snake_pixels) > 0:
            self.pixel_grid[self.snake_pixels.pop()] = 0x000000

    def _new_snake(self):
        """
        Create a new single segment snake. The snake has a random
        direction and location. Turn on the pixel representing the snake.
        """
        # choose a random direction and store it
        self.direction = random.choice(SnakeAnimation.ALL_DIRECTIONS)

        # choose a random starting tile
        starting_tile = (random.randint(0, self.width - 1), random.randint(0,
self.height - 1))

        # add the starting tile to the list of segments
        self.snake_pixels.append(starting_tile)

        # turn on the pixel at the chosen location
        self.pixel_grid[self.snake_pixels[0]] = self.color

    def _can_move(self, direction):
        """
        returns true if the snake can move in the given direction
        """
        # location of the next tile if we would move that direction
        next_tile = tuple(map(sum, zip(
            SnakeAnimation.DIRECTION_OFFSETS[direction], self.snake_pixels[0])))

        # if the tile is one of the snake segments
        if next_tile in self.snake_pixels:
            # can't move there
            return False

        # if the tile is within the bounds of the grid
        if 0 <= next_tile[0] < self.width and 0 <= next_tile[1] < self.height:
            # can move there
            return True

        # return false if any other conditions not met
        return False


    def _choose_direction(self):
        """
        Choose a direction to go in. Could continue in same direction
        as it's already going, or decide to turn to a dirction that
        will allow movement.
        """

        # copy of all directions in a list
        directions_to_check = list(SnakeAnimation.ALL_DIRECTIONS)
```

```
        # if we can move the direction we're currently going
        if self._can_move(self.direction):
            # "flip a coin"
            if random.random() < 0.5:
                # on "heads" we stay going the same direction
                return self.direction

        # loop over the copied list of directions to check
        while len(directions_to_check) > 0:
            # choose a random one from the list and pop it out of the list
            possible_direction = directions_to_check.pop(
                random.randint(0, len(directions_to_check)-1))
            # if we can move the chosen direction
            if self._can_move(possible_direction):
                # return the chosen direction
                return possible_direction

        # if we made it through all directions and couldn't move in any of them
        # then raise the SnakeStuckException
        raise SnakeAnimation.SnakeStuckException


    def draw(self):
        """
        Draw the current frame of the animation
        """
        # if the snake is currently the desired length
        if len(self.snake_pixels) == self.snake_length:
            # remove the last segment from the list and turn it's LED off
            self.pixel_grid[self.snake_pixels.pop()] = 0x000000

        # if the snake is less than the desired length
        # e.g. because we removed one in the previous step
        if len(self.snake_pixels) < self.snake_length:
            # wrap with try to catch the SnakeStuckException
            try:
                # update the direction, could continue straight, or could change
                self.direction = self._choose_direction()

                # the location of the next tile where the head of the snake will
move to
                next_tile = tuple(map(sum, zip(
                    SnakeAnimation.DIRECTION_OFFSETS[self.direction],
self.snake_pixels[0])))

                # insert the next tile at list index 0
                self.snake_pixels.insert(0, next_tile)

                # turn on the LED for the tile
                self.pixel_grid[next_tile] = self.color

            # if the snake exception is caught
            except SnakeAnimation.SnakeStuckException:
                # clear the snake to get rid of the old one
                self._clear_snake()

                # make a new snake
                self._new_snake()

    class SnakeStuckException(RuntimeError):
        """
        Exception indicating the snake is stuck and can't move in any direction
        """
        def __init__(self):
            super().__init__("SnakeStuckException")
```

Conways Game Of Life Animation

The `ConwaysLifeAnimation` is an implementation of Conway's Game of Life (https://adafru.it/KAz). It is a basic simulation of cells that live and die based on simple rules about how many neighbors they have. You get to set the location of a set of live cells at the start of the simulation and then it will step through simulation ticks applying the population rules. During each step the code will iterate over every pixel in the grid and count the number of live and dead neighbor cells, deciding based on the giving rules the fate of the current cell and storing it in a list. After it's made it through the whole grid then it works it's way through the lists created above doing the actual turning on and off of pixels to signify cells spawning and dying.



```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
"""
ConwaysLifeAnimation helper class
"""
from micropython import const

from adafruit_led_animation.animation import Animation
from adafruit_led_animation.grid import PixelGrid, HORIZONTAL


def _is_pixel_off(pixel):
    return pixel[0] == 0 and pixel[1] == 0 and pixel[2] == 0


class ConwaysLifeAnimation(Animation):
    # Constants
    DIRECTION_OFFSETS = [
        (0, 1),
        (0, -1),
        (1, 0),
        (-1, 0),
        (1, 1),
        (-1, 1),
        (1, -1),
        (-1, -1),
    ]
    LIVE = const(0x01)
```

```
    DEAD = const(0x00)

    def __init__(
        self,
        pixel_object,
        speed,
        color,
        width,
        height,
        initial_cells,
        equilibrium_restart=True,
    ):
        """
        Conway's Game of Life implementation. Watch the cells
        live and die based on the classic rules.

        :param pixel_object: The initialised LED object.
        :param float speed: Animation refresh rate in seconds, e.g. ``0.1``.
        :param color: the color to use for live cells
        :param width: the width of the grid
        :param height: the height of the grid
        :param initial_cells: list of initial cells to be live
        :param equilibrium_restart: whether to restart when the simulation gets
stuck unchanging
        """
        super().__init__(pixel_object, speed, color)

        # list to hold which cells are live
        self.drawn_pixels = []

        # store the initial cells
        self.initial_cells = initial_cells

        # PixelGrid helper to access the strand as a 2D grid
        self.pixel_grid = PixelGrid(
            pixel_object, width, height, orientation=HORIZONTAL, alternating=False
        )

        # size of the grid
        self.width = width
        self.height = height

        # equilibrium restart boolean
        self.equilibrium_restart = equilibrium_restart

        # counter to store how many turns since the last change
        self.equilibrium_turns = 0

        # self._init_cells()

    def _is_grid_empty(self):
        """
        Checks if the grid is empty.

        :return: True if there are no live cells, False otherwise
        """
        for y in range(self.height):
            for x in range(self.width):
                if not _is_pixel_off(self.pixel_grid[x, y]):
                    return False

        return True

    def _init_cells(self):
        """
        Turn off all LEDs then turn on ones cooresponding to the initial_cells

        :return: None
        """
```

```
        self.pixel_grid.fill(0x000000)
        for cell in self.initial_cells:
            self.pixel_grid[cell] = self.color

    def _count_neighbors(self, cell):
        """
        Check how many live cell neighbors are found at the given location
        :param cell: the location to check
        :return: the number of live cell neighbors
        """
        neighbors = 0
        for direction in ConwaysLifeAnimation.DIRECTION_OFFSETS:
            try:
                if not _is_pixel_off(
                    self.pixel_grid[cell[0] + direction[0], cell[1] + direction[1]]
                ):
                    neighbors += 1
            except IndexError:
                pass
        return neighbors

    def draw(self):
        # pylint: disable=too-many-branches
        """
        draw the current frame of the animation

        :return: None
        """
        # if there are no live cells
        if self._is_grid_empty():
            # spawn the inital_cells and return
            self._init_cells()
            return

        # list to hold locations to despawn live cells
        despawning_cells = []

        # list to hold locations spawn new live cells
        spawning_cells = []

        # loop over the grid
        for y in range(self.height):
            for x in range(self.width):

                # check and set the current cell type, live or dead
                if _is_pixel_off(self.pixel_grid[x, y]):
                    cur_cell_type = ConwaysLifeAnimation.DEAD
                else:
                    cur_cell_type = ConwaysLifeAnimation.LIVE

                # get a count of the neigbors
                neighbors = self._count_neighbors((x, y))

                # if the current cell is alive
                if cur_cell_type == ConwaysLifeAnimation.LIVE:
                    # if it has fewer than 2 neighbors
                    if neighbors < 2:
                        # add its location to the despawn list
                        despawning_cells.append((x, y))

                    # if it has more than 3 neighbors
                    if neighbors > 3:
                        # add its location to the despawn list
                        despawning_cells.append((x, y))

                # if the current location is not a living cell
                elif cur_cell_type == ConwaysLifeAnimation.DEAD:
                    # if it has exactly 3 neighbors
                    if neighbors == 3:
```

```
                    # add the current location to the spawn list
                    spawning_cells.append((x, y))

        # loop over the despawn locations
        for cell in despawning_cells:
            # turn off LEDs at each location
            self.pixel_grid[cell] = 0x000000

        # loop over the spawn list
        for cell in spawning_cells:
            # turn on LEDs at each location
            self.pixel_grid[cell] = self.color

        # if equilibrium restart mode is enabled
        if self.equilibrium_restart:
            # if there were no cells spawned or despaned this round
            if len(despawning_cells) == 0 and len(spawning_cells) == 0:
                # increment equilibrium turns counter
                self.equilibrium_turns += 1
                # if the counter is 3 or higher
                if self.equilibrium_turns >= 3:
                    # go back to the initial_cells
                    self._init_cells()

                    # reset the turns counter to zero
                    self.equilibrium_turns = 0
```

Both Together

The following **code.py** will run both animations at the same time. By default it expects to find one Neopixel Featherwing (http://adafru.it/2945), and one Dotstar Featherwing (http://adafru.it/3449). You can update the code to use a different configuration of featherwings, or some other compatible grid. Remember to update the pins referenced and initialization as needed.

```
# SPDX-FileCopyrightText: 2024 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
Uses NeoPixel Featherwing connected to D10 and
Dotstar Featherwing connected to D13, and D11.
Update pins as needed for your connections.
"""
import board
import neopixel
import adafruit_dotstar as dotstar
from conways import ConwaysLifeAnimation
from snake import SnakeAnimation

# Update to match the pin connected to your NeoPixels
pixel_pin = board.D10
# Update to match the number of NeoPixels you have connected
pixel_num = 32

# initialize the neopixels featherwing
pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.02, auto_write=False)

# initialize the dotstar featherwing
dots = dotstar.DotStar(board.D13, board.D11, 72, brightness=0.02)

# initial live cells for conways
initial_cells = [
    (2, 1),
    (3, 1),
```

```
    (4, 1),
    (5, 1),
    (6, 1),
]

# initialize the animations
conways = ConwaysLifeAnimation(dots, 0.1, 0xff00ff, 12, 6, initial_cells)

snake = SnakeAnimation(pixels, speed=0.1, color=0xff00ff, width=8, height=4)

while True:
    # call animate to show the next animation frames
    conways.animate()
    snake.animate()
```